Proceedings of BNCOD11, Keele
Springer Verlag LNCS, pages 220-237, 1993

Principles of Implementing Historical Databases in RDBMS

Peter McBrien

Imperial College
Dept. of Computing
180 Queen's Gate
London SW7 2BZ

Tel: +44 71 589 5111
Fax: +44 71 581 8024
Email: pjm@doc.ic.ac.uk

Abstract. The issue of query languages for historical databases has received considerable interest in the database literature over the past decade. Recently temporal relational algebras have been described which provide a theoretical foundation for these languages in the same manner that the relational algebra provides for the SQL language. In this paper the issue of algorithms for the querying and updating of information for one such temporal algebra is discussed, in the specific context of implementing such algorithms on conventional database management systems (DBMS) based on the relational algebra. In so doing, we make apparent the extensions needed to make an RDBMS support any historical database query language with the expressive power of the temporal relational algebra.

Keywords: Historical Databases, RDBMS, Relational Algebra, Query Processing, Temporal Relational Algebra, US Logic

1 Introduction

The topic of adding the temporal dimension to the relational model has received considerable attention in the database literature, a recent survey of which may be found in [1]. Most of the historical databases systems described in the database literature, such as those of [2,13,11,9,6] assume specific methods for the storage of temporal information, and do not address the issue of providing a logic as the basis of the algebra. More recently work has been conducted to develop algebras for historical database querying, based on a conceptual data model called the *temporal structure* [14,5]. These algebras can be related to the *US Logic,* which is widely used for work in the temporal logic area [7,4]. This gives the temporal query languages based on these algebras a theoretical underpinning, allowing for the comparison of their expressive power, and sets a common framework for their implementation independent of the physical data model used.

This paper sets out to explore what are the requirements for changes to current

RDBMS in order that they support query languages based on the *temporal relational algebra* (*TRA*) described in [5]. This will be conducted by showing how the TRA queries may be translated into relational algebra queries on a relational encoding of the temporal structure. The extensions that it is necessary to make to the RA in order that it serves this role form the foundation of the extensions necessary to any RDBMS so that it can support queries of the TRA, or any TRA-based query language.

Section 2 describes some of the previous work which this paper develops upon, and justifies the method by which we choose to store historical data in a relational model. Section 3 describes a set of update primitives for historical data held in a relational model. Section 4 describes algorithms for the full implementation of the TRA in an RA extended with a set of higher-order functions. To give this work a practical perspective, the encoding of the algorithms in the Transact-SQL language of the Sybase RDBMS is presented where the expressive power of the latter makes this possible.

2 Background

The data model used by the TRA is the *temporal structure* introduced in [2], which describes any discrete linear bounded temporal model. In the temporal structure, a historical database $\mathbf{D}_\tau$ (where τ is a natural number), is considered as a set of relational databases $\{ \mathbf{D}_t \mid 0 \le t \le \tau \}$, where the subscript t denotes the time associated with the a particular database. Each $\mathbf{D}_t$ has the same schema as $\mathbf{D}_\tau$, a restriction natural in the context of databases since in practice we would hope to encode the various tuples contained in a relation $\mathbf{R}_t$ over time into a single relation $\mathbf{R}'$. By restricting t to some finite subset of the natural numbers $0 \ldots \tau$ we achieve a bounded model, and using the usual ordering on the natural numbers achieve a linear model. As a shorthand, we will term each instance of t as a *tick.* The structure is extended in [5] to always include in $\mathbf{D}_\tau$ a relation *TIME*, the single attribute of which represents the value of t in any particular $\mathbf{D}_t$.

Example 1 : Example Temporal Structure

For the rest of the paper we shall use the following temporal structure as the database on which example queries will operate. $\mathbf{D}_\tau$ consists of the relations

TIME(T)
EMP(NAME,SALARY)
WORKS_FOR(NAME,MANAGER)

The table below shows tuples of the relations at the first few values of the tick time t

database	*TIME*	*EMP*	*WORKS_FOR*
$\mathbf{D_0}$	(0)	(Peter,100) (Richard,90) (Ed,200)	(Peter,Ed) (Richard,Ed)
$\mathbf{D_1}$	(1)	(Peter,110) (Richard,90) (Ed,200) (Dov,200)	(Peter,Ed) (Richard,Dov)
$\mathbf{D_2}$	(2)	(Peter,110) (Richard,140) (Ed,200) (Dov,200)	(Peter,Dov) (Richard,Dov)
$\mathbf{D_3}$	(3)	(Peter,140) (Richard,140) (Dov,200)	(Peter,Dov) (Richard,Dov)
$\mathbf{D_4}$	(4)	(Peter,140) (Richard,140) (Ed,200) (Dov,200)	(Peter,Dov) (Richard,Dov)

As a shorthand, we may write tuples of a relation which hold at several ticks as the tuple followed by the set of ticks at which the tuple holds. For example, we may write the various instances of (Dov,200) being a member of *EMP* as (Dov,200) {1,2,3,4}. For a continuous series of ticks this set can be further simplified to an interval, which for the example results in (Dov,200) [1,4].

It should be noted that by associating real times with both the length of each tick and the time of tick 0, we may easily convert tick numbers to actual time values. For instance, if we choose a tick to be one minute, and tick 0 to be 11am 1st March 1991, then we know tick 120 is 1pm 1st March 1991, tick 1440 is 11am 2nd March 1991, *etc*.

The temporal structure may be queried using the *temporal relational algebra* (TRA), which consists of variants of the five RA operators, namely select σ, project π, Cartesian product $\times$, set difference $-$ and union $\cup$, together with the temporal operators *since-product* $S_\times$ and *until-product* $U_\times$. The semantics of the operators is defined by the following table, where q is the database tick time of the query.

TRA	*Result for* $\mathbf{D}_q$
$R\ \mathrm{S}_\times\ S$	The product of all tuples from S at $\mathbf{D}_s$ with those from R at all $\mathbf{D}_t$ for which $s \le t < q$
$R\ \mathrm{U}_\times\ S$	The product of all tuples from S at $\mathbf{D}_s$ with those from R at all $\mathbf{D}_t$ for which $q < t \le s$
$R \cup_t S$	The union of tuples from R at $\mathbf{D}_q$ with those from S at $\mathbf{D}_q$
$R \times_t S$	The product of tuples from R at $\mathbf{D}_q$ with those from S at $\mathbf{D}_q$
$R -_t S$	The difference of tuples in R at $\mathbf{D}_q$ with those in S at $\mathbf{D}_q$
$\pi_t\ R$	The projection of tuples from R at $\mathbf{D}_q$
$\sigma_t\ R$	The selection of tuples from R at $\mathbf{D}_q$

The evaluation of a *query* in the TRA can be made with respect to a database tick time q, in which case we write '*query* at q'. Alternatively the query can be made with respect to all times, the result for each $\mathbf{D}_q$ being '*query* at q'. Note that the five operators with names the same as the RA operators only produce output relations from input relations at the same $\mathbf{D}_q$ — that is taking the snapshot view of each time, the results produced by the operators are identical to those produced the RA operators. The new temporal operators produce output tuples in the 'current' database $\mathbf{D}_q$ from input relations in 'other' databases $\mathbf{D}_t$ (where $t \neq q$).

Example 2 : Some Results of TRA Queries on the Temporal Structure of Example 1

db	*EMP* $\mathrm{S}_\times$ *TIME(0)*	π_{NAME} *EMP*	*EMP* $\mathrm{S}_{\bowtie}$ *WORKS_FOR*
$\mathbf{D_0}$	$\varnothing$	(Peter) (Richard) (Ed)	$\varnothing$
$\mathbf{D_1}$	(Peter,100) (Richard,90) (Ed,200)	(Peter) (Richard) (Ed) (Dov)	(Peter,100,Peter,Ed) (Richard,90,Richard,Ed)
$\mathbf{D_2}$	(Richard,90) (Ed,200)	(Peter) (Richard) (Ed) (Dov)	(Peter,110,Peter,Ed) (Richard,90,Richard,Ed) (Richard,90,Richard,Dov)
$\mathbf{D_3}$	(Ed,200)	(Peter) (Richard) (Dov)	(Peter,110,Peter,Ed) (Peter,110,Peter,Dov) (Richard,140,Richard,Dov)
$\mathbf{D_4}$	$\varnothing$	(Peter) (Richard) (Ed) (Dov)	(Peter,140,Peter,Dov) (Richard,140,Richard,Dov)

Example 2 gives a table of results for each $\mathbf{D}_q$ of Example 1 for the following TRA queries:

- *EMP* $S_{\times}$ *TIME*(*0*) gives at each $\mathbf{D}_q$ the tuples of *EMP* which have always held in the past.

- π_{NAME} *EMP* gives for each $\mathbf{D}_q$ the projection of the *NAME* attribute of *EMP*.

- *EMP* $S_{\bowtie}$ *WORKS_FOR* gives the temporal natural *since-join* on the *NAME* attribute of *EMP* and *WORKS_FOR*, where

$$R\ S_{\bowtie}\ S = \pi_{i1,i2,...,im}\ \sigma_{R.A1=S.A1,...,R.An=S.An}\ (R\ S_{\times}\ S),$$

  and A_1, A_2, ..., A_n are the shared attributes of R and S. This will return tuples which have held at all times in the past in the *EMP* relation since there was one in the *WORKS_FOR* relation with the same instance of the *NAME* attribute.

In [5] a method for storing and querying the temporal structure in a standard RDB was described, the resulting system being termed a *temporal relational database* (*TDB*). In short, it proposed that tuples $(a_1,...,a_n)$ $[start,end]$ of relation R are stored in an RDB as $R(a_1,...,a_n,start,end)$. To find if a tuple holds for the temporal structure at time q we only need ask the RDB query $\sigma_{start \leq q \leq end}\ R(a_1,...,a_n,start,end)$.

Adding the constraint that all such intervals are maximal gives the *temporal normal form* (*TNF*) of [9]. A fundamental premise of this paper is that all intervals stored in the RDB are maximal, a corollary of this is that for any tuple $R(a_1,...,a_n,start,end)$ there must be no other *matching* tuple $R(a_1,...,a_n,start',end')$ where the non-temporal attributes $a_1,...,a_n$ are identical, and the temporal attributes *overlap* by conforming to $start \leq end'+1$ and $end \geq start'-1$ (strictly speaking we are finding intervals which overlap or just touch each other). Also, to be a valid interval in the bounded model $0 \leq start \leq end \leq \tau$ must be satisfied. The effect of this restriction is that if we see $R(a_1,...,a_n,start,end)$ stored in the DB, then we know both $(a_1,...,a_n)[start-1]$ and $(a_1,...,a_n)[end+1]$ to be false. Since the notion of two intervals overlapping will be repeatedly referred to in the following discussions and definitions, we define a function here to determine if *overlap* holds for a pair of intervals.

function overlap([*start*,*end*],[*start′*,*end′*])
    if $start \leq end'+1$ **and** $end \geq start'-1$ **then return true**
    else return false

The justification for choosing the TNF, against allowing overlapping intervals to be stored in the RDB, can be summarized as follows:

- It produces a more compact database, since the overlapping intervals of two tuples are replaced by a single interval for a single tuple.

- If the normal form were not chosen, then the primary keys of relations would be seen to have duplicates at some times. For example, if we have a relation *R* with a single attribute which is also its primary key, and we have stored in the database *R*= { (1) [2,10] , (1) [10,20] }, then at tick 10 the primary key 1 is seen to be duplicated.

- The implementation of the query language would be complicated if we had to cater for an essentially continuous piece temporal of information being 'broken' across several tuples. For instance, if we want to find if *R*(1) held for the interval [5,15], then in the TNF database one needs only check that *R*(1) is stored in the database as holding for some interval which encloses [5,15], but in the non-TNF database we must merge all overlapping intervals to see if together they enclose [5,15].

Example 3 illustrates how the temporal structure may be encoded in relational form (the intervals are of course stored simply as two extra attributes of the *EMP* and *WORKS_FOR* relations).

Example 3 : TNF encoding of Example 1

EMP	*WORKS_FOR*
(Peter,100)[0,0]	(Peter,Ed)[0,1]
(Peter,110)[1,2]	(Peter,Dov)[2,4]
(Peter,140)[3,4]	(Richard,Ed)[0,0]
(Richard,90)[0,1]	(Richard,Dov)[1,2]
(Richard,140)[2,4]	
(Ed,200)[0,2]	
(Ed,200)[4,4]	
(Dov,200)[1,4]	

Obviously the operation of inserting or deleting tuples requires we maintain the TNF. This will complicate the implementation of these operations, since when inserting information we must now first check if this can be achieved by extending the interval of an existing tuple, and when deleting we must check if the tuple still holds for other ticks. This matter will be dealt with in Section 3.

In [5] the TRA operators were partially implemented in the RA by performing arithmetic on the intervals of tuples. However, due to a lack of expressive power in the RA only a limited implementation could be achieved in this manner. Section 4 will describe an extension of the implementation which completely implements the TRA in the RA extended with a set of higher-order functions. This serves to make apparent what enhancements are necessary to RDBMS to fully implement query processing languages with the expressive power of TRA, one example being the Temporal-SQL.

3 Storing and Updating a Historical Database in an RDB

The encoding of the temporal structure using intervals has an obvious implementation in an RDBMS, where we need only add two extra attributes to a relation to store the intervals that each tuple is valid for. Using the Sybase Transact-SQL [12] as an example implementation language, if we have a relation *R* with a single integer attribute, then we need add `start_time` and `end_time` attributes to store the time stamp, to achieve the following definition of the relation:

```
CREATE TABLE R (
       arg1         int      not null,
       start_time   int      not null,
       end_time     int      not null)
```

We now shall consider how insertions to, and deletions from, the relations held in such a schema may be achieved.

3.1 Insertions into the TDB

To maintain the requirement that intervals stored in the RDB are maximal, we must not simply add new tuples to the database, but merge the tuple with any matching tuples in the database which have intervals overlapping. The addition of a tuple t_{new} = $(a_1,\dots,a_n)$ $[start,end]$ of relation *R* may be achieved by the following procedure:

procedure insertTDB($R,(a_1,\dots,a_n)$ $[start,end]$)

(i) S = { $(a'_1,\dots,a'_n)$ $[start',end']$ **for_which** $(a'_1,\dots,a'_n)$ $[start',end'] \in R$ **and** $a_1=a'_1, \dots, a_n=a'_n$ **and** overlap($[start,end],[start',end']$) **}**

(ii) **for_all** $t \in S$ **delete** t **from** R

(iii) $S' = S \cup (a_1,\dots,a_n)$ $[start,end]$

(iv) $start_{min}$ = **minimum** { *start* **for_which** $[start,\_] \in S'$ }
end_{max} = **maximum** { *end* **for_which** $[\_,end] \in S'$ }

(v) **insert** $(a_1,\dots,a_n)$ $[start_{min},end_{max}]$ **into** R

Step (i) finds all tuples already stored in the relation that would violate TNF if the t_{new} were added. These are any tuples with the same attributes as the t_{new} and an associated interval which overlaps that of t_{new}. In step (ii) all these violating tuples are deleted, then in step (iii) t_{new} is added to the set so that we may compute in (iv) the maximum interval covering existing tuples and t_{new}. Step (v) then inserts a new tuple into *R* with the attributes of t_{new} and an interval extended to cover the period of existing tuples with the same attributes as t_{new}.

To implement this as a stored procedure in the Sybase Transact-SQL language is straight forward, but unfortunately the language does not allow a variable to hold a relation name (one may not say `SELECT * FROM @R` where `@R` is a variable), and thus we can only present here the format that the procedure can taken by giving an example for a specific relation *R* with a single integer attribute arg_1. The procedure begins by using a single `SELECT` statement to find the matching tuples in the database which overlap the one being added, and finding their minimum start, and maximum end time. If no such tuples are found we may add the new tuple in the database, otherwise we merge in the interval of the new tuple with the overlapping intervals just found, delete all the overlapping tuples, and then insert the new tuple. Note that it would be more efficient if we were able to delete the overlapping tuples as we found them with the initial `SELECT`, but this is not possible in Transact-SQL

```
CREATE PROCEDURE insertTDB_R
   @data1 int,
   @start int,
   @end int  AS

DECLARE @P int
DECLARE @Q int

SELECT @P=MIN(start_time),@Q=MAX(end_time)
FROM R
WHERE arg1=@data1 AND start_time<=@end+1
             AND end_time>=@start-1

IF @P IS NULL
   BEGIN
   INSERT INTO R VALUES (@data1,@start,@end)
   END
ELSE
   BEGIN
   IF (@P>@start) SELECT @P=@start
   IF (@Q<@end) SELECT @Q=@end
   DELETE FROM R
   WHERE arg1=@data1 AND start_time>=@P AND end_time<=@Q
   INSERT INTO R VALUES (@data1,@P,@Q)
   END
```

3.2 Deletions from the TDB

To remove a tuple over some interval of time requires that we *clip* all matching tuples in the RDB which have overlapping intervals. It fact we may narrow the overlapping criterion for two intervals [*start,end*] and [*start′*,*end′*] to $start \leq end'$ and $end \geq start'$ since we are not interested in touching intervals (they could not clip each other). The removal of a tuple $t_{new} = (a_1,...,a_n)$ [*start,end*] of relation *R* may be achieved by the following procedure:

procedure removeTDB(R,t_{new})

(i) S = { $(a'_1,...,a'_n)$ [$start',end'$] **for_which** $(a'_1,...,a'_n)$ [$start',end'$] $\in R$ **and** $a_1=a'_1, \ldots , a_n=a'_n$ **and** $start \leq end'$ and $end \geq start'$) **}**

(ii) **for_all** $t \in S$ **delete** t **from** R

(iii) $S' = S \cup (a_1,...,a_n)$ [$start,end$]

(iv) $start_{min}$ = **minimum** { $start$ **for_which** [$start$,_] $\in S'$ }
end_{max} = **maximum** { end **for_which** [_,end] $\in S'$ }

(v) **if** $start_{min}$ < start **then insert** $(a_1,...,a_n)$ [$start_{min},start-1$] **into** R
if $end_{max} > end$ **then insert** $(a_1,...,a_n)$ [$end+1,end_{max}$] **into** R

Save for the slight optimisation of the overlapping criterion, this algorithm is identical in steps (i) to (iv) to that for *insertTDB*. The only difference is in step (v), where we insert into the database the tuple for the times outside the interval specified for deletion, which is the complement of the interval we found in step (v) of the *insertTDB* algorithm.

As was the case for the insertion operation, the implementation of this procedure in Transact-SQL requires a separate procedure to be provided for each relation in the database. As an example of the form such procedures can take, the stored procedure definition for removing temporal information from a single argument relation R is given below.

```
CREATE PROCEDURE removeTDB_R
   @data1 int,
   @start int,
   @end   int  AS

DECLARE @P int
DECLARE @Q int

SELECT @P=MIN(start_time),@Q=MAX(end_time)
FROM R
WHERE arg1=@data1 AND start_time<=@end
             AND end_time>=@start

IF @P IS NOT NULL
   BEGIN
   DELETE FROM R
   WHERE arg1=@data1 AND start_time>=@P
             AND end_time<=@Q
   IF (@P<@start)
       INSERT INTO R VALUES (@data1,@P,@start-1)
```

```
IF (@Q>@end)
    INSERT INTO R VALUES (@data1,@end+1,@Q)
END
```

4 Querying the TDB

In this section we detail how the TRA operators may be implemented in terms of RA operators operating on the TDB encoding of the temporal structure, with the RA extended with two new higher-order functions.

4.1 Implementing the TRA in the RA

When making a query on the TDB, we expect an answer in the TNF. If this were not the case we might get incorrect results when the output relation from the query was as an input for another TRA operator, since each operator will be implemented to work on a database in the normal form. Relations are stored in TNF, so to ensure the result of a query is in TNF we need only consider that the implementation of all TRA operators results in an output relation which is in TNF. Each operator of the TRA will be given a definition in terms of a RA query on the TDB described in Section 1, and an estimate of the complexity of the execution of the operator will be made.

Select

Since a *select* serves only to reduce the number of tuples in a relation, the output from a *select* must be in TNF if the input relation *R* is. Thus we may use the RA *select* to implement directly the TRA *select*, and the complexity is of the order of the number of tuples in *R*.

$$\sigma_{t\ \mathrm{A},\ldots}\ R = \sigma_{\ \mathrm{A},\ldots}\ R$$

$$complexity(\sigma_t\ R) = O(N)\ \mathbf{where}\ N = \mathbf{count}(R)$$

Project

In implementing *project*, we meet a problem concerning the maintenance of the TNF in the output from the operator, due to the *project* making what were distinct tuples at overlapping times become identical tuples at overlapping times, and thus violating the TNF. For example, the projection of the *NAME* attribute from the *EMP* relation if implemented using the RA *project* operator would give the following result:

$$\pi_{NAME}\ EMP = \text{(Peter) [0,0]}\quad \text{(Peter) [1,2]}\quad \text{(Peter) [3,3]}$$

which should be in TNF just (Peter) [0,3]. If we are to use the output from the RA *project* as the result of the TRA *project*, we need to define a higher-order function *coalesce* which removes such violations of the TNF. This may be achieved by sorting the relation on its attributes, and then merging all tuples which have the same attributes and overlapping intervals. A pseudo code algorithm for *coalesce* is defined below, which functions to filter the tuples returned by a sort of the relation to see if any two adjoining tuples in the sorted relation $t_1=(a_1,...,a_n)$ $[start,end]$ and $t_2=(a'_1,...,a'_n)$ $[start',end']$ satisfy $a_1=a'_1, \ldots, a_n=a'_n$, $start' \leq end+1$, in which case they should be replaced by a single tuple $t_{12}=(a_1,...,a_n)$ $[start,max(end,end')]$

function coalesce(R)
(i) R := R **order_by** $A_1,...,A_n$,START_TIME
$R_C = \varnothing$
$size$ = **count**(R)
i = 1
(ii) **repeat**
$(a_1,...,a_n)$ $[start,end] = R_i$
while $(a'_1,...,a'_n)$ $[start',end'] = R_{i+1}$ **and** $a_1=a'_1, \ldots, a_n=a'_n$
and $start' \leq end+1$
end := max(end',end)
i := i+1
end_while
insert $(a_1,...,a_n)$ $[start,end]$ **into** R_C
i := i+1
while $i \leq size$
(iii) **return** R_C

The construct $(a_1,...,a_n)$ $[start,end] = R_i$ in the above definition finds the ith tuple of relation R. Using the coalesce function we may define the TRA *product* in terms of the coalesce function being applied to the output of the RA *project*. The complexity of the TRA project is thus of order $N \log(N)$, where N is the number of tuples in R, since the complexity of the coalesce function is $O(N \log(N))$ due to the use of a sort in the function definition.

$$\pi_{t\,A,B,C}\, R = \text{coalesce}(\pi_{A,B,C,START\_TIME,END\_TIME}\, R)$$

$$complexity(\pi_t\, R) = O(N \log(N)) \quad \textbf{where } N = \textbf{count}(R)$$

It is interesting to note that an operator called *contract,* with the same semantics as *coalesce,* was provided for in Sarda's algebra [10]. In [10] it was asserted that the operator was needed if the proposed algebra was to be considered expressively complete. This statement seems to be validated here by the need to introduce the operator in order to support the implementation of the TRA in the RA.

Product

From the definition of the TRA *product* operator, we see that the product of two tuples holds for all $\mathbf{D}_q$ in the temporal structure where both the tuples hold. Thus for the *product* to hold in the TNF, the intervals of the two tuples must overlap, and the period for which the product holds starts at the maximum of the start times of the two tuples, and ends at the minimum of the end times for each tuple. Providing the following functions to compute those *max* and *min* allows a definition of the TRA product to be given in terms of the RA *product* operator. The TRA *product* thus has the same complexity as the RA *product*.

function min(x,y)
  if $x \leq y$ **then return** x
  else return y

function max(x,y)
  if $x \geq y$ **then return** x
  else return y

$$R \times_t S \ = \ \pi_{R,S,\max(R.\text{START\_TIME},S.\text{START\_TIME}),\min(R.\text{END\_TIME},S.\text{END\_TIME})} \left(\sigma_{\text{overlap}(R,S)} (R \times S) \right)$$

complexity$(R \times_t S) = O(NM)$ **where** N = **count**(R) **and** M = **count**(S)

Union

As was the case with *project*, the simple use of the RA *union* would result in an output relation violating TNF, since bringing two relations together can make a tuple from one relation overlap a relation in another. Thus we must use the *coalesce* function to merge the possibly overlapping intervals from the two relations.

$$R \cup_t S \ = \ \text{coalesce}(R \cup S)$$

complexity$(R \cup_t S) = O(N \log(N))$ **where** N = **count**(R)+**count**(S)

Difference

A peculiarity of using intervals for the encoding of the temporal structure is that the difference operator $R - S$ may result in more tuples than are present in R. For example, the query {(Dov,200) [1,4]} – {(Dov,200) [2,3]} results in two output tuples {(Dov,200) [1,1] , (Dov,200) [4,4]}. This leads to a re-implementation the relational *difference* operator, so that it reduces or splits the size of the interval R holds for as it searches the relation S. This procedure if defined by the following pseudo-code program.

function difference(R,S)=$RS\_$

(i) $RS\_ := \varnothing$

(ii) $\forall\ (a_1,\ldots,a_n)\ [start,end] \in R$
do $interval = \{\ [start,end]\ \}$
$\forall\ (a_1,\ldots,a_n)\ [start',end'] \in S$ **do** $interval := interval - [start',end']$
$\forall\ [start'',end''] \in S$ **do** $RS\_ := RS\_ \cup (a_1,\ldots,a_n)\ [start'',end'']$

(iii) **return** $RS\_$

The function definition takes each tuple of R in turn, builds a set of intervals for which that tuple holds by iterating over tuples from S which overlap it. This set is initialized with the interval associated with the tuple in R, and has each of the intervals for which the same tuple is found in S removed. This generates a set of intervals for which the tuple is in R but not S, each of which is added to the output $RS\_$.

$R -_t S$ = difference(R,S)
$complexity(R -_t S) = O(NM)$ **where** N = **count**(R) **and** M = **count**(S)

Since-Product

From the definition of *since-product* working on the temporal structure given in Section 1, it can be seen that the following axioms hold for the operator when working on the encoding of the temporal structure as intervals (the axioms for *until-product* are the obvious 'reflections' in time):

(i) The tuples of $R\ \mathrm{S}_\times\ S$ hold for at least the interval $[s_{RS}+1, e_{RS}+1]$, where $[s_{RS}, e_{RS}]$ is the interval that the tuples of $R \times S$ hold, due to the fact for any $\mathbf{D}_q$ in the temporal structure, $R\ \mathrm{S}_\times\ S$ holds for all tuples that $R \times S$ holds for in $\mathbf{D}_{q-1}$.

(ii) The tuples of $R\ \mathrm{S}_\times\ S$ hold for the interval $[s_{RS}+1, e_R+1]$ when $s_R \leq e_{RS} \leq e_R$ where $[s_R, e_R]$ is the interval that the tuples of R hold, due to the fact that $R\ \mathrm{S}_\times\ S$ always holds in $\mathbf{D}_q$ for all tuples that both R and $R\ \mathrm{S}_\times\ S$ hold for in $\mathbf{D}_{q-1}$.

Figure 1 illustrates the result on the second axiom diagrammatically for a few of the tuples taken from Example 1. Clearly all that we need to evaluate a query $R\ \mathrm{S}_\times\ S$ is to find the intervals over which R and S hold, then use the axioms listed above to create the interval for the *since-product*. The interval for axiom (i) is computed by a simple modification to that used to generate the intervals for $\times_t$, and then axiom (ii) is taken into account by making the end of the interval be that of R, not the maximum of R and S.

$$R\ \mathrm{S}_\times\ S = \mathrm{coalesce}(\pi_{R,S,\mathrm{max(R.START\_TIME},S\mathrm{.START\_TIME)+1},R\mathrm{.END\_TIME+1}}\ \sigma_{\mathrm{overlap}(R,S)}\ (R \times S)\ )$$

$$complexity(R\ \mathrm{S}_\times\ S) = O(NM + (N+M)log(N+M))$$

$$\textbf{where } N = \textbf{count}(R) \textbf{ and } M = \textbf{count}(S)$$

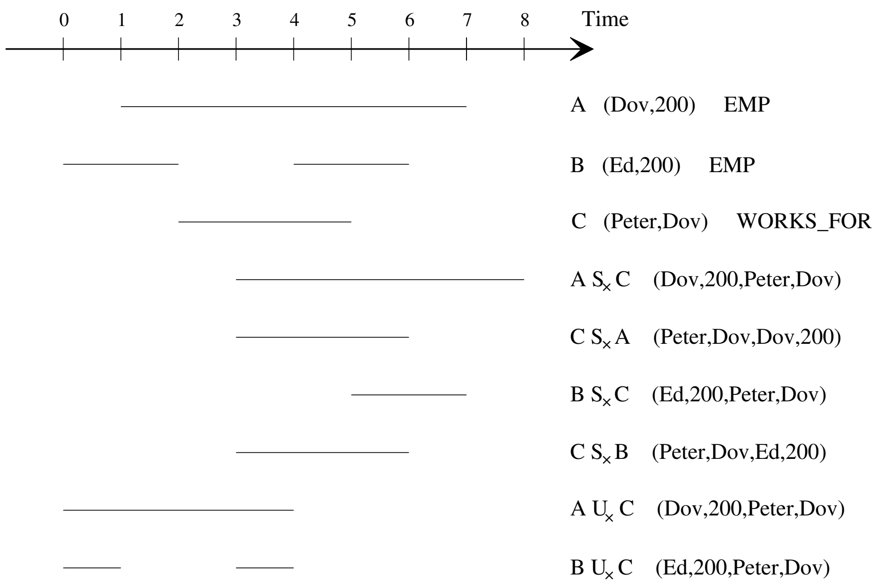

Figure 1 : Examples of intervals for which tuples hold

This use of the coalesce function is only necessary to remove 'extra' tuples of the form $(a_1,...,a_n)$ [start,end] where there is another tuple of the form $(a_1,...,a_n)$ [$start'$,end'] for which $start' \leq start \leq end \leq end'$. An example will make this point clear. In Figure 1 we illustrate some results of the *since-product* and *until-product* when we have *EMP* = { (Dov,200) [1,7], (Ed,200) [0,2], (Ed,200) [4,6] }, and *WORKS_FOR* = { (Peter,Dov) [2,5] }. Evaluating *WORKS_FOR* $\mathrm{S}_\times$ *EMP* using the above algorithm will return the tuples { (Peter,Dov,Ed,200) [3,6] , (Peter,Dov,Ed,200) [5,6] , (Peter,Dov,Dov,200) [4,8] } if no coalesce function is used. The tuple (Peter,Dov,Ed,200) is seen to have an 'extra' result as holding over [5,6]. Since the correct (maximal interval) result is also returned it may be realized that this is a problem of presentation. In a practical query language it would be sensible to allow the user to specify when the coalesce function is used, in the same manner that one can specify the use of `DISTINCT` in an SQL `SELECT` to remove duplicates [3].

Until-Product

From the definitions of the *since-product* and *until-product* it may be seen the latter operator is merely the 'reflection' of the former, working into the future instead of the past. We thus need only give the definition of the encoding here, using as justification the commentary given for the *since-product.*

$$R \ \mathrm{U}_{\times} \ S = \mathrm{coalesce}(\pi_{R,S,R.\mathrm{START\_TIME}-1,\min(R.\mathrm{END\_TIME},S.\mathrm{END\_TIME})-1} \ \sigma_{\mathrm{overlap}(R,S)} (R \times S) \ )$$

$$\mathit{complexity}(R \ \mathrm{U}_{\times} \ S) = O(NM + (N+M)\mathit{log}(N+M)) \quad \textbf{where } N = \textbf{count}(R) \textbf{ and } M = \textbf{count}(S)$$

4.2 Optimisations

The algorithms presented in the previous subsection make no attempt at optimizing the queries. It is not intended to cover this topic in any depth within this paper, but to indicate the directions in which future work may proceed. In very broad terms, we may divide the issues of query optimisation of the TRA queries into two categories:

(i) **Structure of Query**

The query may be the subject of some static analysis to determine the extent to which the TNF must be maintained in the output of each TRA operator that is a constituent of the query. For example:

- If a query is made with respect to a specific tick time, then we do not need to maintain the TNF in the output relation from principal operator of the query. Thus if we ask $\pi_t \ R$ at time(t) we can simplify the implementation of the project to $\pi \ \sigma_{\mathrm{START\_TIME} \le t \le \mathrm{END\_TIME}} \ R$.
- The coalesce function used in the definition of $\mathrm{S}_{\times}$ is only needed if the output relation from the operator is not in the scope of another operator that uses *coalesce*. This is because the output of $\mathrm{U}_{\times}$ and $\mathrm{S}_{\times}$ without the application of *coalesce* only violates TNF by producing tuples with an interval which is a sub-interval of some other tuple. Thus they only violate the TNF by making some tuples appear to be duplicated are certain times, which can be ignored except for purposes of presenting results (or using aggregate operators such as

SUM which we do not consider here).

- The schema design should help to avoid the use of projects in queries, since the execution of projects in the TRA now has $O(N\,log(N))$ complexity on the number of tuples in the relation.

- We can use the fact that the non-temporal operators can accept input in non-TNF, producing semantically correct answers as output, but still in non-TNF form. As can be seen from the table below, this allows us to use the RA implementation of an operator, excluding the use of coalesce for the majority of operations. Indeed, only operators that require TNF as input (i.e. $S_\times$ and $U_\times$), and have the input produced by an operator which does not produce TNF (i.e. $S_\times$, $U_\times$, $\cup_t$ and π_t), need have coalesce applied.

TRA	*non-TNF input?*	*Implement in RA?*
$S_\times$	✗	Yes (but produces 'subset' results)
$U_\times$	✗	Yes (but produces 'subset' results)
$\cup_t$	✓	Yes (use *coalesce* to get TNF)
$\times_t$	✓	Yes
$-_t$	✓	No (entirely new algorithm used)
π_t	✓	Yes (use *coalesce* to get TNF)
σ_t	✓	Yes

(ii) **Query Execution**

The specification of the algorithms for coalesce, difference *etc* has been made external to the definitions of the algorithms for the RA operators. Since operators like product, since-product and until-product will be used to make a join, which will involve some sorting of the tuples from each relation involved, it is apparent that the TRA would much more efficiently implemented if these algorithms were more integrated.

(iii) **Physical Data Model**

Although we have used the data model of [9] as our implementation vehicle, this does not preclude the use of many different physical data models for the storage of the temporal information, which may give benefits in compactness when handling large amounts of information. The only requirement would be that the query processing be able to readily abstract the TNF form from the stored data without significant overhead. For instance, a common technique for overcoming the redundancy in the variation of a single attribute in a large tuple is to use *delta tuples*, where one records only the attribute that has changed together with its time-stamp. The results of this technique for the *EMP* relation of Example 3 is shown in Example 4. Clearly a database system supporting such delta tuples may be easily made to support the abstraction of 'full' tuples without great implementation overhead.

Example 4 : Delta tuple form of the *EMP* relation of Example 3

Δ *EMP*
(Peter,100)[0,0]
Δ(110)[1,2]
Δ(140)[3,4]
(Richard,90)[0,1]
Δ(140)[2,4]
(Ed,200)[0,2]
(Ed,200)[4,4]
(Dov,200)[1,4]

Summary & Conclusions

This paper has presented methods by which a historical database may be stored and updated in a standard relational model, and queries of the TRA translated into an extended RA working on the relational model. The complexity of the TRA operators has been shown not to be significantly greater than that of their RA counterparts, with the exception of the TRA project which has complexity $O(N\, log(N))$ on the number of tuples in a relation. Design of database schemas to avoid the use of projects thus becomes important in the context of historical databases.

It has been shown that extending the RA with the *coalesce* and *difference* functions is all that is required in order to fully implement the TRA in the RA. By extending a RDBMS with these primitives we would allow for the implementation of any historical database query language with the expressive power of the TRA.

The techniques presented in this paper are currently being used as the foundation for the implementation of a Datalog-like temporal query language in the Tempora Esprit project, using the Sybase DBMS as the implementation platform. Future work will need to consider in greater depth the areas of query optimisation, looking at both static analysis, and at the integration of the algorithms detailed in this paper with existing techniques for the dynamic optimisation of RA queries, such as fast join algorithms. Future work will also consider how the techniques described may be extended to handle aggregate functions.

Acknowledgements

The work reported in this paper was partly funded by the EC under Esprit project number 2469. The author would like to thank all members of the project for their helpful involvement in numerous meetings, and in particular the contributions of M.Finger, D.Gabbay and R.Owens at Imperial College, and A.Conti, M.Niézette, F.Schumacker and P.Wolper at the University of Liège. The author would also like to thank M.Finger and R.Owens for much helpful advice during the preparation of this paper.

References

1. J.F.Allen: *Maintaining Knowledge about Temporal Intervals,* CACM Vol. 26, No. 11 pp 832-843, 1983.
2. J.Clifford & A.U.Tansel, *On An Algebra for Historical Relational Databases: Two Views*, Proc. ACM SIGMOD Conference 1985.
3. C.J.Date, *A Guide to the SQL Standard*, Addison-Wesley, 1989.
4. D.Gabbay, *The Declarative Past and Imperative Future*, Temporal Logic in Specification: Altrincham Workshop 1987, LNCS 398 pp 409-448, Springer-Verlag, 1989.
5. D.Gabbay & P.J.McBrien, *Temporal Logic and Historical Databases,* Proceedings of the 17th International Conference on Very Large Databases, Barcelona, 1991.
6. S.K.Gadia, *A Homogeneous Relational Model and Query Languages for Temporal Databases*, ACM TODS, Vol. 13, No. 4 pp 418-448, 1988.
7. H.Kamp, *On the Tense Logic and the Theory of Order,* PHD Thesis, UCLA, 1968.
8. L.E.McKenzie Jr. & R.T.Snodgrass, *Evaluation of Relational Algebra Incorporating the Time Dimension in Databases,* ACM Computing Surveys, Vol.23 No.4, December 1991.
9. S.B.Navathe & R.Ahmed, *TSQL - A Language Interface for History Databases*, Temporal Aspects of Information Systems, pp 109-122, *Ed.* C.Rolland, F.Bodart & M.Leonard, North-Holland, 1988.
10. N.Sarda, *Algebra and Query Language for a Historical Data Model.* Computing Journal, Vol 33, No. 1 pp 11-18, 1990.
11. R.Snodgrass, *The Temporal Query Language TQuel,* ACM TODS Vol. 12, No. 2 pp 247-298, 1987.
12. Sybase, *Transact-SQL User's Guide,* 9th October, 1989, Sybase Inc.
13. A.U.Tansel, *Adding Time Dimension to Relational Model and Extending Relational Algebra*, Information Systems Vol 11, No. 4, pp 343-355, 1986.
14. A.Tuzhilin & J.Clifford*, A Temporal Relational Algebra as a Basis for Temporal Relational Completeness*, Proceedings of the 16th International Conference on Very Large Databases, Brisbane, 1990.